MADELEINE: An Efficient and Portable Communication Interface for RPC-Based Multithreaded Environments

Luc Bougé, Jean-François Méhaut, Raymond Namyst

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MADELEINE : An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments

Luc Bougé
LIP, ENS Lyon

Jean-François Méhaut
LIFL, Univ. Lille I

Raymond Namyst
LIP, ENS Lyon

No 3459

July 1998

THÈME 1

ISSN 0249-6399

RHÔNE-ALPES

MADELEINE : An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments

Luc Bougé *
LIP, ENS Lyon

Jean-François Méhaut†
LIFL, Univ. Lille I

Raymond Namyst*
LIP, ENS Lyon

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n ° 3459 — July 1998 — 15 pages

Abstract: Due to their ever-growing success in the development of distributed applications, today's multithreaded environments have to be highly portable and efficient on a large variety of hardware. Most of these environments have an implementation built on top of standard communication interfaces such as PVM or MPI, which are widely available on existing architectures. Obviously, this approach ensures a high level of portability. However, we show in this paper that these communication interfaces do not meet the needs of RPC-based multithreaded environments as far as efficiency is concerned. The contribution of this paper is to propose a new portable and efficient communication interface, called MADELEINE, that is especially designed for such multithreaded environments. We report on several implementations of MADELEINE on top of various network protocols that demonstrate the efficiency of our approach.

Citation. An extended abstract of this report will be published under the following reference: *Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98), ENST, Paris, France, October 1998. To appear.*

Key-words: High-Performance Computing. Distributed Multithreading. RPC-based communications. High-Speed Networks and Workstation Clusters. Zero-Copy Protocols. PM2 Programming Environment.

(Résumé : tsvp)

* Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon, F-69364 Lyon, France. Email : {Luc.Bouge,Raymond.Namyst}@ens-lyon.fr

† Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, F-59655 Villeneuve d'Ascq Cedex, France. Email : Jean-Francois.Mehaut@lifl.fr

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
T´el´ephone : 04 76 61 52 00 - International: +33 4 76 61 52 00
T´el´ecopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

MADELEINE : Une interface de communication portable et efficace pour les environnements multithreads à base de RPC.

Résumé : Compte tenu de leur utilisation croissante pour le développement d'applications distribuées, les environnements multithreads actuels se doivent d'être à la fois portables et efficaces sur un vaste éventail d'architectures matérielles. L'implantation de la plupart de ces environnements s'appuie d'ailleurs sur des interfaces de communication de haut niveau telles que PVM ou MPI. Cependant, nous montrons dans cet article que ces interfaces ne conviennent pas aux environnements communiquant sur la base de RPC, en particulier en ce qui concerne l'efficacité. Cet article propose donc une nouvelle interface de communication portable et efficace, nommée MADELEINE, mieux adaptée aux besoins de ces environnements. Nous décrivons l'implantation de cette interface sur plusieurs protocoles réseaux et montrons que cette approche est très efficace.

Citation. Une version abrégée de ce rapport va être publiée sous la référence suivante. *Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine : an efficient and portable communication interface for multithreaded environments. In Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98), ENST, Paris, France, October 1998. To appear.*

Mots-clé : Calcul hautes performances. Multithreading distribué. Communications basées sur les RPC. Reseaux haut debit et grappes de stations. Protocoles zéro-copie. Environnement de programmation PM2.

1 Introduction

Distributed multithreading is currently a very popular technique to support the execution of parallel applications on MIMD machines. This is mainly due to the recent emergence of distributed multithreaded environments [13, 15, 20] and to their availability on a wide range of architectures. The distributed multithreaded approach has already been successfuly applied to several application fields, such as Combinatorial Optimization [9], Data Parallel Compiling [23], Large Simulations [4] and Molecular Dynamics [1].

For portability reasons, multithreaded environments are often implemented on top of high-level standard communication interfaces such as PVM or MPI. A large subset of these environments are RPC-based environments because the provided functionalities rely implicitly or explicitly on a mechanism able to invoque remote services (*e.g.* RSR in Nexus [13], RPC in PM2 [20], Fork/Join in DTS [5]).

In this article, we show that existing communication interfaces do not adequately support the implementation of these environments, either because they are not portable (too low-level) or simply because they do not match the specific needs of these environments. Usually, implementations fall into the second category because most of existing environments emphasize portability (*e.g.* Chant [15], Athapascan [3]).

This situation has often been considered as a "natural" tradeoff between portability and efficiency. We prove that this idea is wrong and we present a new portable communication interface (called MADELEINE) that provides to upper level software both transparency and high efficiency for RPC-based multithreaded environments. The MADELEINE software is organized in two layers. The low-level layer (portability) isolates the code that needs to be adapted for each targeted network protocol. The high-level layer (API) provides communication functionalities including buffer management features. Thanks to the small number of architecture-dependent primitives, MADELEINE is straightforward to implement on top of many network protocols, including very low level ones, such as BIP [24] or SBP [25]. In this respect, it has similarities with the Illinois FM communication software.

The remainder of this article is organized as follows. In the next section, we introduce the specific needs of RPC-based MTE as far as communication is concerned. Section 3 briefly presents the most representative existing communication interfaces and emphasizes why they do not match the needs described in the previous section. The contribution of our article is presented in section 4 where the MADELEINE communication interface is described and discussed. Section 5 describes the implementation of MADELEINE on top of several network protocols and shows some performance results. Finally, section 6 summarizes the contribution of this work and give the main points we intend to address in the near future.

2 What Communication Interface do RPC-based Multithreaded Environments Need ?

A multithreaded environment is called "RPC-based" if most communications take place by means of remote invocations of services (also called Remote Procedure Calls). In this scheme of remote interaction, a client (usually a thread) sends a request to a server (usually a process) which executes a specified function (a service) to serve the request. According to the type of service executed, a response may be sent back to the client upon completion of the function. Many kinds of remote operations actually conform to this communication scheme. For instance, remote-read and remote-write functionalities that are provided by some distributed environments [11] are RPC operations. Thread migration [20, 7] is another example, where the parameter of the service is basically the thread's stack and where no response is expected on the calling side. We observe that most existing multithreaded environments provide RPC-based mechanisms [13, 15, 3, 5].

The following subsections present three main requirements that today's multithreaded environments should meet : zero-copy data transmissions, control over communication scheduling and portability. While the last two ones are not specific to RPC-based environments, the first requirement has a dramatic influence on the efficiency of RPC operations.

2.1 Towards zero-copy data transmissions

On a high speed network providing very low transmission latency (*e.g.* Myrinet [2]), any extra message copy introduced at some software level has a tremendous impact on performance [18]. Therefore, it is crucial

that the underlying communication layer be able to ensure the delivery of messages with no intermediate extra copy of data. Many existing communication libraries actually provide such a functionality, including those providing a very low level of network abstraction [27, 22, 24, 25] but also those – such as MPI – providing a rich set of high-level communication functionalities.

However, the problem of realizing an RPC operation without any extra copy of data is more complex than realizing a zero-copy message transmission. Actually, when a RPC request is sent to a server, the server does not know the location where the data should be placed until it extracts preliminary information from the request message. Typically, the triggering of a RPC operation takes place in the following three steps on the server side :

1. The request type is identified (*i.e.* "Is this a migration or a remote put ?") and some information is extracted from the network.
2. Then, actions are taken (for instance dynamic memory allocations) to prepare for the receipt of the data.
3. Finally, the data are extracted from the network and stored directly at the right location in memory.

As a result, implementing this scheme while avoiding extra copies of messages will require the client to send two messages[1] : the first one to carry the "*header*" of the request and the second one to carry the regular data (the "*body*"). However, this approach does not yield good results if the underlying communication software is still buffering data on the receiving side. In this case, a single message would probably have been sufficient to carry all the information (header + body), since it could have been extracted in several steps. We will note in Section 3 that most existing communication interfaces do not allow the construction of programs that behave "optimally" in both situations. That is, the programmer has to adapt his program according to the underlying networking protocol used, which is not acceptable in terms of portability.

2.2 Control over communication scheduling

Multithreaded applications are often very sensitive to the policy used to schedule communication operations, especially if the network is accessed in user-level space. This is commonly the case with optimized interfaces to high-speed networks such as BIP [24], SBP [25] or U-Net [26]. In this case, the application is neither signaled by the system nor by the hardware when a network event (such as the receipt of a packet) occurs. Instead, the application has to explicitly poll the network for incoming events. As there is a tradeoff between the application responsiveness and the overhead of useless polling actions [13], a multithreaded environment should be able to tune its polling frequency.

Furthermore, a thread waiting for the completion of a network operation (by polling for the appropriate event) should block and yield the processor to another thread to overlap communication with computation. This is only possible at the multithreaded environment level, since communication layers are usually not "thread aware". A communication interface for multithreaded environments should therefore provide explicit polling operations.

2.3 Portability

Roughly speaking, the implementation of a distributed multithreaded environment essentially realizes the complex integration of a thread package and communication software. In order to be portable on a wide range of architectures, these two components must themselves be portable. As far as threads are concerned, the problem is not really so critical since almost all thread packages provide approximately the same functionalities. This makes it easy to build a common interface (actually similar to the POSIX-thread interface [17]) which can be adapted to the targeted thread package in a straightforward manner.

In contrast, the situation is much more complex as far as communication is concerned. Due to the variety of today's networking technologies (Ethernet, ATM, Myrinet, SCI), efficient communication libraries often provide low-level interfaces focusing on a particular network technology. As a result, their implementations are highly optimized for the underlying network hardware (BIP, SBP, SHMEM). However, building complex applications (such as multithreaded environments) directly on top of such an interface would lead to rewriting

[1] With a classical message passing interface.

large parts of them when porting to other network hardware. This is the reason why many people got involved in the design of portable communication interfaces (PVM [14], MPI [19]) that hide the network dependent features (reliability, flow control, message fragmentation, etc.) by providing a high level of abstraction for the network.

3 What Do Existing Communication Interfaces provide ?

Many communication libraries have been recently designed to provide portable interfaces and/or efficient implementations to build distributed applications. They essentially fall into two classes. *High-level* communication libraries hide all underlying network features and provide advanced facilities for the exchange of data between computers (message passing [19], tuple space [6]). *Low-level* communication libraries aim at getting the maximum performance out of the underlying hardware.

We now review some representative communication libraries in each class and discuss how they match the criteria described in the previous section.

3.1 High-level Interfaces

PVM (Parallel Virtual Machine) is certainly one of the most popular "de facto standard" high-level communication libraries of the last decade. This is mainly due to its portability and ease of use, rather than to its performance. On this point indeed, PVM is surpassed by communication interfaces able to avoid buffering of messages, such as MPI. Some multithreaded environments were implemented on top of PVM (PM2 [20], DTS [5] and TPVM [12]), but their performance on high-speed networks was often poor[2] because of the numerous extra copies of messages made by the library.

MPI (Message Passing Interface), which is the standard communication interface proposed by the MPI Forum (academic + vendors), is not subject to several PVM limitations. Its implementations usually ensure the transmitting of messages without extra copies. However, this implies that the receiver knows (in advance) the type and size of the data it will get. This is clearly not the case in RPC-based environments, as we discussed in Section 2.1. Chant [15] and Athapascan [3], two multithreaded environments built on top of MPI, suffer from this drawback.

3.2 Low-level Interfaces

Because high-level interfaces are not always able to meet the needs of time-critical applications, several research teams have designed low-level communication interfaces that can deliver much of the underlying hardware's performance. Usually, these libraries are used as a basis for the building of high-level communication libraries (MPI-FM [18]).

Although most of these libraries are message-passing oriented, they provide different interfaces and various levels of quality of service (reliability, flow-control, etc.). For instance, BIP (Basic Interface for Parallelism [24]), which is today's most efficient way to transmit data over a Myrinet network, allows arbitrarily long messages to be sent but does not provide reliability or flow-control. SBP (Streamlined Buffer Protocol [25]) and U-Net [26], which are highly optimized communication layers for Ethernet and Fast-Ethernet networks, provide reliability[3] and flow-control but force the upper layers to deal with preallocated fixed-sized buffers for message manipulation. These few examples help to explain why an implementation of a multithreaded environment directly built on top of these communication interfaces would not be portable. The implementation of the Nexus multithreaded environment (Argonne, USA [13]) is based, as far as communications are concerned, on a generic interface that could adapt to these low-level libraries. However, this interface does not provide sufficiently fine control over the underlying network protocols, so that buffering of messages is still done on receiving nodes.

The FM (Illinois Fast Messages) communication interface [22] is derived from the AM (Active Messages) communication software designed at Berkeley [27]. It could be considered as a "medium-level" communication layer in that sense that it can be ported on top of previously mentioned low-level ones. Its interface provides

[2]Except for vendor specific versions such as PVMe on the Cray T3D.

[3]Only SBP actually provides reliability.

a very simple mechanism to send data to a receiving node that is notified upon arrival by the activation of a handler. The 2.0 release of this interface provides some interesting gather/scatter features thanks to the introduction of a "streaming message" concept that allows the sending and the receiving of data in several (not necessarily equal) pieces. This feature may allow an efficient implementation of zero-copy RPC operations as presented in section 2.1. However, this streaming functionality is too general. Thus, the efficiency would not be optimal on networks such as Myrinet, where each segment would need a separate acknowledgment to enforce flow-control. The communication interface we propose in the next section does not have this drawback and provides a higher level of abstraction to the upper layers.

4 The MADELEINE Communication Interface

To meet the needs discussed in the previous sections, we propose a new communication interface — called MADELEINE — especially designed for RPC-based multithreaded environments. The main characteristic of this interface [21] is that it is based on dynamic cooperation with the upper software layers (*i.e.* the multithreaded environment) to avoid extra copies of transmitted data. At the lowest level, this cooperation is realized using up-calls on the receiving side to allow upper layers to interactively participate in data extraction. At the highest level (*i.e.* above MADELEINE), this cooperation is elegantly accomplished through the use of simple buffer management primitives.

The structure of MADELEINE is actually composed of two software layers : a low-level portability interface and a high-level generic programming interface, as show in Figure 1.

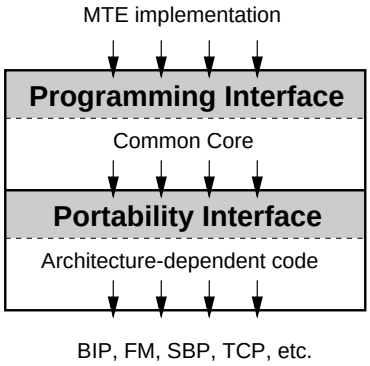

FIG. 1: Structure of the MADELEINE communication interface

4.1 The Portability Layer

The *Portability Layer* of MADELEINE is intended to provide a common interface to the various communication subsystems on which it may be ported. The design of this interface is very critical because it has to be independent of any particular protocol while staying efficiently portable on top of any of them.

We chose an execution model that is inspired by the Active Messages model [27], with some extensions and changes that probably make it closer to Fast Messages [22]. Thus, our model is essentially a message-passing protocol where exchanges are done between traditional processes. This means that although its implementation has to be thread-safe, the protocol does not need to be thread-aware. As a consequence, upper layers have to ensure that only one thread is processing a receive operation at a time.

Messages consist of a set of one or more vectors. Each vector is a contiguous area of memory and is defined by a pair (*address*, *size*) where the size is expressed in bytes. The first vector of a message has particular semantics with respect to the receive operation : this vector contains the data that must be extracted prior to the rest of the message (*i.e.* the "header"). Once extracted, these data are immediately made available to the upper layers so that some application-level actions may be executed before the rest of the message is extracted.

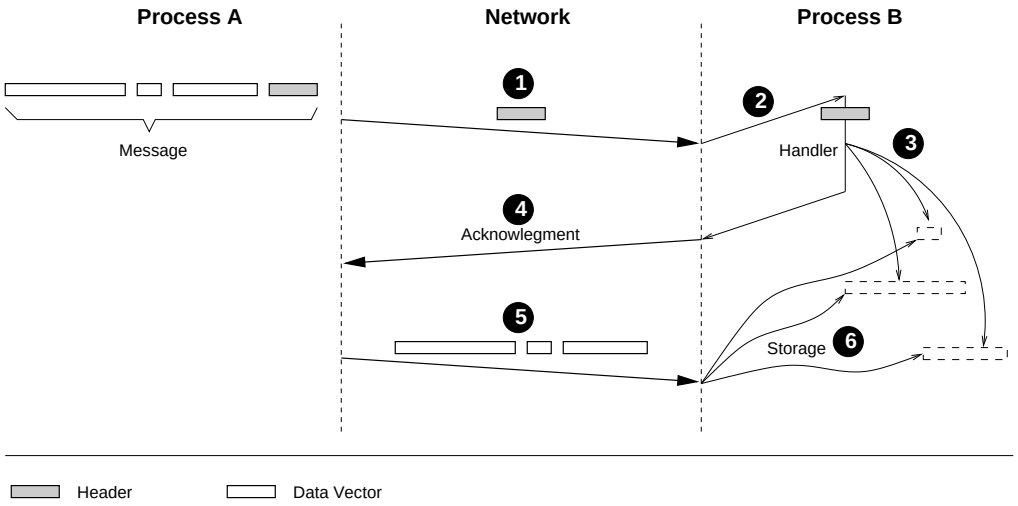

FIG. 2: Conceptual view of the data-exchange protocol in the portability layer.

The Figure 2 sketches a situation where process A is sending a message to process B. First, the message *header* (*i.e.* the first vector) is sent to process B. On its receipt, a *handler* is executed with the header as a parameter. This handler, which is defined by the upper layer, can inspect the header and take appropriate actions to prepare for full message extraction. Typically, the handler may compute the locations where data should be placed (step (3) in Figure 2). Process A is allowed to send the vectors of remaining data (i.e. the message body) when the handler has completed. Consequently, these data are directly stored at the right memory locations on their receipt.

We emphasize that this scenario is a conceptual description of the protocol. In fact, specific implementations may not strictly follow this communication scheme. We discuss more precisely the implementation issues in Section 5.

4.1.1 The Portability Interface

The portability interface we propose is composed of only 8 function prototypes (Table 1) which represent the architecture-dependent primitives of MADELEINE. Two of them (`init` and `exit`) deal with the management of connections respectively at the beginning and at the end of the execution of an application. The remaining 6 primitives deal with communication.

TAB. 1: The MADELEINE portability interface.

Function	Operation
`init(configSize, taskIDs)`	Connections Setup
`exit()`	Connections Shutdown
`send_post(dest, iovec, count)`	Post the sending of msg to node 'dest'
`send_poll(dest)`	Check if msg sending is completed
`recv_header_post(func)`	Ready to receive a header
`recv_header_poll()`	Check if header received
`recv_body_post(exped, iovec, count)`	Ready to receive data
`recv_body_poll(exped)`	Check if data received

4.1.2 Message sending

As MADELEINE is intended to be used in a multithreaded context, operations that would ordinarily block the calling process (*e.g.* the sending of a message) should only block the calling thread. However, MADELEINE

has no knowledge about the thread package which is actually used in the upper layers. Furthermore, the control over thread scheduling (and thus over communication scheduling) should be done by these upper layers to better meet the application needs (see Section 2.2).

Therefore, a sending operation is done in two steps. A call to the `send_post` primitive initiates the sending of a message and returns without blocking. The destination process and the message (an array of Unix standard `iovec` structures) are given as parameters. Then, further calls to `send_poll` have to be made until the primitive returns TRUE, which means that message transmission has completed on the sending side.

4.1.3 Message receiving

During the receipt of a message, the first step is to receive the message header, which is done by a call to `recv_header_post(handler)` followed by one or more calls to `recv_header_poll()`. As soon as the header is completely received, the call-back function "*handler*" gets executed. Thus, the execution flow returns temporarily back to the upper layers.

The idea is that the handler has to inspect the header, to prepare for the receipt of the message body and to actually extract the data from the network. The second step typically leads to building an appropriate array of vectors. The last step is realized by making a call to `recv_body_post` (followed by subsequent calls to `recv_body_poll`) with the array as a parameter.

4.2 The Programming Interface

Although the portability layer offers a network-independent communication interface, it is still too rudimentary to be accessed "as is" by the upper layers, *i.e.* by a multithreaded environment. In particular, the management of the arrays of vectors that are needed by several primitives is a low-level job that should be kept inside MADELEINE.

The goal of the Programming Interface is to provide a set of convenient primitives for the management of messages. Basically, this interface provides *packing* and *unpacking* primitives that allow the upper layer to specify how data should be inserted in/extracted from messages. Of course, this interface must clearly provide a way to distinguish between *header data* and *regular data*, because the associated semantics are completely different.

In addition, these facilities have to be available at multiple levels of the upper software, to allow the use of piggybacking techniques without losing efficiency. The following example illustrates this need. Consider a remote procedure call which takes an unbounded array as a parameter. When the request reaches the destination node, the header is examined both by the multithreaded runtime (to allocate the appropriate thread stack and then to spawn the server thread) and by the user application (to allocate the memory where the array should be stored). This shows that several software layers may actually participate in the packing/unpacking of message data.

The MADELEINE programming interface is essentially composed of a set of *packing* and *unpacking* primitives. A packing primitive simply appends some data to a message under construction. An unpacking primitive does the dual operation by extracting some data from a given message. The prototypes of these functions are very similar to the buffer management primitives provided by the PVM interface (*e.g.* `pvm_pkint`, etc.). For instance, the function used to append one or more integers to the current message has the following prototype :

```
void pack_int(int mode, int *elems, int nb);
```

The *mode* parameter plays an important role in the MADELEINE interface, because it determines the semantics of the operation. It can be assigned three different values. A *packing* operation may behave differently according to this mode :

`IN_HEADER` forces MADELEINE to put the data into the "header" of the message, so that they are guaranteed to be available as soon as the message reaches the destination node.

`IN_PLACE` specifies that MADELEINE should do its best to transmit the corresponding data without using extra-copies. Furthermore, the data will be actually read from memory when the send operation is

executed. This means that any modification of these data between their packing and their sending will actually update the message contents.

`BY_COPY` should be used to allow further modifications to the data without impacting on the message contents. It is provided for convenience only, since it is always possible for an application to deal with the appropriate data copies and to use the `IN_PLACE` mode.

On the receiving side, *unpacking* operations should specify the same mode as the corresponding packing one. As discussed above, `IN_HEADER` ensures that data is immediately readable after the unpacking. On the contrary, the user should consider that data extracted either `IN_PLACE` or `BY_COPY` is only readable after the receipt of the message is completed. In MADELEINE, this takes place on the return of the *handler* (see Figure 2).

4.2.1 Example

Figure 3 illustrates the typical use of high-level MADELEINE primitives to receive a message and extract data from it. In this example, the message contains a byte-array whose size was stored in the header of the message.

```
1   int size;
2   char *data;
3
4   void handler()
5   {
6      unpack_int(IN_HEADER, &size, 1);
7                                /* 'size' is available immediately */
8      data = malloc(size);
9      unpack_byte(IN_PLACE, data, size);
10                    /* Caution: data is *not* yet available here. */
11  }
12
13  int main_function()
14  {
15     ...
16     on_receipt_call(handler);             /* Handler registration */
17     while(!msg_received())
18        pthread_yield();     /* Yields execution to another thread */
19                         /* Here, a message has been fully received */
20     printf("I received the msg: %s\n", data);
21  }
```

FIG. 3: Receiving and inspecting a message with the MADELEINE Programming Interface

First, the receiving process registers a *handler* (Line 16). This handler will be executed as soon as a message arrival is detected. If a message is already available, the handler is executed immediately. If not, polling on the `msg_received` operation has to be done periodically (Lines 17 and 18). As soon as a message is available, the handler gets executed. When it finishes, the `msg_received` primitive returns `TRUE`. In the handler, the first unpacking operation is done to extract the size of the array (Line 6). Note that the mode was set to `IN_HEADER`, so that the size is immediately usable and actually gets used as a parameter of the `malloc` primitive (Line 8). The second unpacking operation is performed to extract the array (Line 9). However, because the mode was set to `IN_PLACE`, the MADELEINE runtime will defer the effective operation until the end of the handler. This way, all `IN_PLACE` unpacking operations will be gathered by MADELEINE and optimally processed in a single step. Thus, the data can be safely accessed only after the handler execution has completed, that is, after `msg_received` returns `TRUE` (Line 19).

5 Implementation and Performance

As described above, the MADELEINE interface has been ported on a variety of low-level communication layers including BIP/Myrinet [24], SBP/Fast-Ethernet [25], Dolphin-SCI-driver/SCI [10] and TCP/IP. It has also been ported on top of PVM [14] and MPI [19], since some vendors implementations of these libraries are very efficient (*e.g.* MPI-F on IBM SP2 or PVMe on Cray T3D). Among this set of low-level communication layers, we distinguish two categories : those which do provide buffering (such as TCP) and those which do not (such as BIP).

5.1 MADELEINE over buffering protocols : TCP

We call buffering protocols the protocols which store data in a temporary place on the destination side of a message transmission. Thus, effective receipt of these data can be deferred, and messages can be read in several steps. Of course, strict zero-copy data transmissions are not possible with these protocols. The TCP/IP transport protocol belongs to this category, as well as protocols such as SBP/Fast-Ethernet.

The implementation of MADELEINE over TCP, which provides flow control, is straightforward. The sending primitives of the portability layer (see Table 1) are just implemented by calls to the non-blocking `writev` Unix primitive. The message (header + body) is thus simply sent as a stream of bytes which is regulated by the internal TCP flow control algorithm.

On the receiving side, the detection of the message availability (`recv_header_poll`) is realized through a call to the Unix `select` primitive. Then, the message header is extracted by means of a `read` operation. Finally, the message body can be extracted with a `readv` operation. As a result, no extra message nor extra copy is introduced by MADELEINE on top of TCP.

5.1.1 Performance

To illustrate the adequacy of MADELEINE for RPC-based environments, we have measured the performance of processing remote procedure calls respectively over MADELEINE and over MPI. The parameter of the procedure call is an array of bytes which is transmitted without any extra copy (except inside TCP itself of course). This experiment has been done on a pile of PCs (Intel PentiumPro 200MHz with 64MBytes of memory running Linux) interconnected by a Myrinet network.

As discussed previously (Section 2.1), the only way of realizing such a remote invocation with MPI is to send two messages. The first one carries the header (containing the procedure Id and the array size) and the second one transports the array itself. With MADELEINE, we used the `IN_HEADER` mode to pack the array size and the `IN_PLACE` mode to pack the array itself (as in the example shown in Figure 3). Figure 4 reports the times measured from the initialization of the communication on the source machine to the effective beginning of procedure execution on the destination machine. These times correspond to various array sizes as indicated on the abscissa.

As one can see, the gap in performance between MADELEINE and MPI is huge and proportional to message size (times obtained with MPI are approximately twice the times obtained with MADELEINE). This shows the better adequacy of the MADELEINE interface to support RPC operations on buffering protocols. Here, the performance gap is due to the number of TCP messages exchanged with the MPI version. In fact, MADELEINE needs only one TCP message to process the RPC, whereas the MPI interface requires the application to send two "user" messages. In addition, the internal flow-control algorithm of the MPI implementation may even trigger the sending of a third message from the receiver back to the sender. It is important to note that this gap is not in any way due to the "quality" of the MPI implementation. It simply reveals that the MPI interface lacks functionalities as far as RPC-like operations are concerned. Of course, experiments featuring other buffering protocols (such as SBP [25]) lead to similar results.

5.1.2 Overhead

Although the advantage of using MADELEINE to build RPC-based multithreaded environment is obvious, it is important to verify that the overhead introduced by MADELEINE is small and independent of the

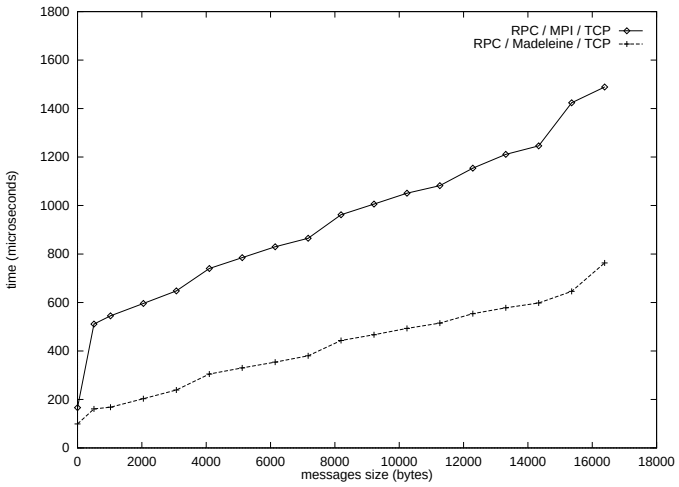

FIG. 4: Performance of Remote Procedure Calls over MADELEINE and over MPI. The underlying communication protocol is TCP/IP.

message length. In particular, this property guarantees that the maximum communication bandwith reached by MADELEINE is close to that of the underlying hardware.

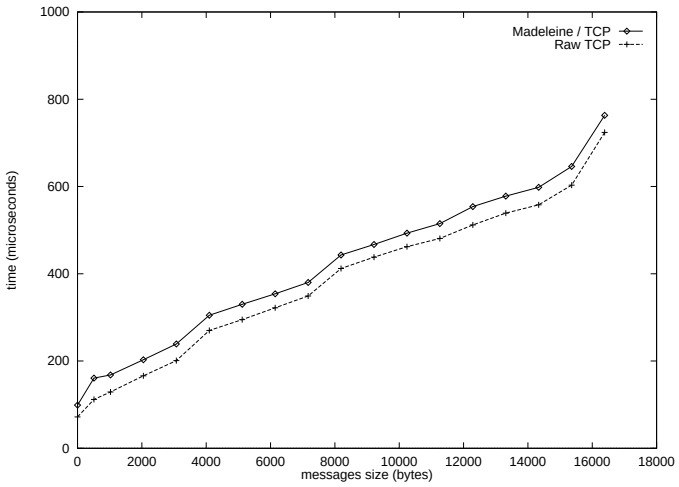

FIG. 5: Overhead of MADELEINE on TCP/IP.

Figure 5 shows the performance of raw message passing over TCP compared with the performance of RPC on top of MADELEINE. Although the comparison between remote procedure calls and raw message transfer is not really fair (see Section 2.1), the two curves show that the overhead is actually constant and small ($30\mu s$). Since only one TCP message is involved during a MADELEINE remote procedure call, this overhead is attributable to the few additional data added into the header and obviously to the software overhead of MADELEINE.

5.2 MADELEINE over zero-copy protocols : BIP

Communication protocols which can realize zero-copy transmissions obviously do not provide internal buffering of data. This means that a message should actually be sent only when the destination node is ready to receive it. Thus, a flow-control mechanism has to be set up between the sender and the receiver. Some

communication protocols integrate such a mechanism internally, but some others leave this responsibility to upper layers.

BIP (Basic Interface for Parallelism [24]) is a very low-level communication interface dedicated to the Myrinet network. The major feature of BIP is to allow zero-copy message transmission between end-point processes. This is realized by providing access to the Network Interface Card (NIC) in user space. Thus, the operating system is not involved in communications and the performance of this communication layer is really impressive (latency of $4.4\mu s$ for 4-bytes messages, and maximum bandwidth of 126 MB/s). BIP provides no flow-control for messages of arbitrary length, but small messages may be sent even if the corresponding receive operation has not yet been posted. In this case, they are temporarily stored in a fixed-size buffer, but no flow-control is done to avoid overflowing this buffer.

The implementation of MADELEINE over BIP is more complex than over TCP, because of flow-control. In particular, a MADELEINE message transmission usually requires three BIP messages, as sketched in Figure 2. The first BIP message carries the header, the second informs the sender that the receiver is ready, and the last one carries the data. Since the BIP interface does not yet allow the sending of non-contiguous data in a single message, more than three messages will be sent if the MADELEINE message is composed of several data vectors. To avoid buffer overflow when sending (small) messages carrying the MADELEINE header, a credit-based flow-control algorithm was implemented.

All the primitives of the portability layer interface are directly implemented on top of BIP asynchronous operations, which fit exactly the semantics of MADELEINE's *post/poll* operations. Furthermore, because BIP uses the DMA engine of the Myrinet card to do the transfers between the computer's main memory and the NIC, send and receive operations can be highly overlapped with computations.

5.2.1 Performance

We have done the same experiment as the one presented in Section 5.1.1 to evaluate the gain of using MADELEINE for the implementation of RPC operations in a multithreaded context. In fact, the MADELEINE and the MPI test programs remain unchanged. Only the MADELEINE and the MPI implementations were changed. As far as MPI is concerned, we used an optimized implementation of MPICH [28] over BIP that was realized by the BIP research team. The results are shown in Figure 6.

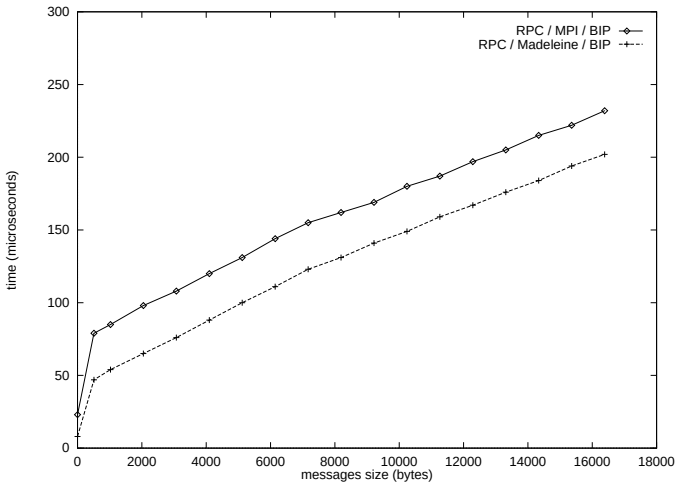

FIG. 6: Performance of Remote Procedure Calls over MADELEINE and over MPI. The underlying communication protocol is BIP.

As one can see, the gap between MADELEINE and MPI is constant and not so large. This result was expected because the MADELEINE implementation on top of BIP does approximately the same job as the implementation of MPICH : three BIP messages are used to accomplish a single remote procedure call (as described above). Nevertheless, MADELEINE is noticeably better than MPI. In fact, the small overhead of the

MPI application is due to the implementation of MPICH itself whose layered design introduces a considerable software overhead compared to MADELEINE.

5.2.2 Overhead

Unlike the TCP implementation, the BIP implementation of MADELEINE introduces an overhead due to several factors :

- First, some additional data are transmitted, such as the tag of the request or some piggybacked flow-control fields (four or five 32-bit words in practice).
- Second, additional messages may be transmitted either to carry the previously mentioned data or to ensure flow control.
- Last, MADELEINE obviously introduces some software overhead to manage messages and handlers.

In fact, this overhead is independant of the size of the data. Furthermore, it is less than $15\mu s$ on our pile of PCs.

Note however that small messages avoid much of this overhead, because MADELEINE is usually able to save two messages (out of three) by putting all the data (header + body) into a single message. Of course, the size of these messages is limited to a fixed number of bytes in order to fit into preallocated buffers on the receiving side. Under these conditions, the overhead of MADELEINE becomes very low, and the latency measured for a null-RPC (*i.e.* with no argument) using MADELEINE is only $8\mu s$ on top of BIP. This is even better than raw message sending on top of MPICH/BIP.

5.3 Discussion

The previous experiments illustrate that the implementation of a RPC-based environment should not rely on traditional message-passing communication layers. Indeed, their interface is not adequate for this purpose. Thus, even if their implementation is very efficient for a given network technology, applications that need RPC facilities may not be able to achieve much of the underlying hardware's performance.

Our experiments have demonstrated the adequacy of MADELEINE for RPC-based environments as far as efficiency is concerned. In particular, the overhead introduced by MADELEINE over the underlying communication subsystem is small and constant. This means that the maximum network bandwidth achieved with MADELEINE is very close to that of the underlying layer. Over BIP, for instance, MADELEINE achieves more than 99% of the underlying layer's bandwidth when the amount of data transmitted exceeds 200 KBytes.

6 Conclusion and future work

Most existing multithreaded environments provide RPC-based functionalities such as invocation of remote services, remote memory accesses or thread migration. However, their implementation is based on message-passing communication interfaces such as PVM or MPI for portability reasons. We have shown that this approach is not efficient on several network protocols because a classical message passing interface lacks expressiveness as far as RPC operations are concerned.

We have proposed a new portable communication interface which better meets the needs of these environments. We have described the two layers that compose MADELEINE. A programming interface provides high-level functionalities to upper-level software, and this layer is based on a portability interface which hides the network specifics. Both these layers consider messages as two-part entities, and attribute different semantics to each part. We have explained how this structure allows efficient (zero-copy) data transmissions during RPC operations.

We have implemented MADELEINE on top of several low-level network interfaces and have reported its performance on top of TCP/IP and BIP. These experiments have demonstrated the superiority of MADELEINE over classical message-passing interfaces. They also have proved that MADELEINE can be easily implemented on low-level protocols while achieving much of the underlying hardware's performance. MADELEINE is currently available on top of the following network interfaces : TCP, BIP, SBP, Dolphin-SCI, PVM and MPI.

The implementation of an existing multithreaded environment (PM^2) has been successfully ported on top of MADELEINE. The performance of PM^2 on top of MADELEINE validates our work : with a BIP-based implementation on a PentiumPro 200MHz cluster connected by Myrinet, a null RPC takes $11\mu s$ and a thread migration takes $80\mu s$.

In the near future, we intend to port MADELEINE on CRAY's SHMEM [8] and on the IBM SP2 interconnection network [16]. We also plan to extend the MADELEINE portability interface in order to optimally exploit network protocols providing fixed-size preallocated buffers, such as SBP [25]. Finally, we are currently working on a multi-protocol extension of MADELEINE which will allow the management of heterogeneous network configurations.

Références

[1] BERNARD, P.-E., AND TRYSTRAM, D. Report on a Parallel Molecular Dynamics Implementation. In *Parallel Computing* (Bonn, Germany, May 1997).

[2] BODEN, N., COHEN, D., FELDERMANN, R., KULAWIK, A., SEITZ, C., AND SU, W. Myrinet : A Gigabit per second Local Area Network. *IEEE-Micro 15-1* (feb 1995), 29–36. Available from *http ://www.myri.com/research/publications/Hot.ps*.

[3] BRIAT, J., GINZBURG, I., PASIN, M., AND PLATEAU, B. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference* (Passau, Germany, 1997), Springer Verlag, pp. 590–599.

[4] BRUNETT, S., AND GOTTSCHALK, T. Scalable ModSAF Simulations with more 50,000 Vehicles using Multiple Scalable Parallel Processors. Tech. Rep. CACR-156, Center for Advanced Computing Research, California Institute of Technology, December 1997.

[5] BUBECK, T., AND ROSENSTIEL, W. Verteiltes Rechnen mit DTS (Distributed Thread System). In *Proc. '94 SIPAR-Workshop on Parallel and Distributed Computing* (Suisse, Oct. 1994), M. Aguilar, Ed., Fribourg, pp. 65–68.

[6] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Communication of ACM 32*, 4 (April 1989), 444–458.

[7] CASAS, J., KONURU, R., OTTO, S., PROUTY, R., AND WALPOLE, J. Adaptative load migration systems for PVM. In *Proceedings of Supercomputing* (Washington D.C., November 1994), ACM/IEEE, pp. 390–399.

[8] CRAY RESEARCH, SGI. *Application Programmer's Library Reference Manual*, 1997. SR-2165.

[9] DENNEULIN, Y., NAMYST, R., AND MEHAUT, J. Architecture Virtualization with Mobile Threads. In *ParCo'97 (PARallel COmputing)* (Sep 1997), Elsevier Science Publishers.

[10] DOLPHIN INTERCONNECT SOLUTIONS INC. *PCI-SCI Adapter Programming Specification*, March 1997.

[11] FERRARI, A., AND SUNDERAM, V. TPVM. A Threads-Based Interface and Subsystem for PVM. Tech. Rep. CSTR-940802, University of Virginia & Emory University, USA, August 1994.

[12] FERRARI, A., AND SUNDERAM, V. TPVM : Distributed Concurrent Computing with Lightweight Processes. In *Proc. of IEEE High Performance Distributed Computing* (Washington, 1995), D.C., pp. 211–218.

[13] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37 (1996), 70–82.

[14] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., AND SUNDERAM, V. *PVM : Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[15] HAINES, M., CRONK, D., AND MEHROTRA, P. On the design of chant : A talking threads package. In *Proc. of Supercomputing'94* (Washington, November 1994), pp. 350–359.

[16] IBM CORP. *The IBM RS/6000 SP High-Performance Communication Network*. available on Mktools Spswp1.

[17] IEEE STANDARDS DEPARTMENT. *1003.4d8 POSIX System Application Program Interface : Threads Extensions [C language]*, 1994.

[18] LAURIA, M., AND CHIEN, A. MPI-FM : High performance MPI on workstation clusters. *Jounal on Parallel and Distributed Computing*, 40 (01) (1997), 4–18.

[19] MESSAGE PASSING INTERFACE FORUM. *MPI : A Message-Passing Interface Standard*, March 1994. available from www.mpi-forum.org.

[20] NAMYST, R., AND MEHAUT, J. PM^2 : Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)* (Sep 1995), Elsevier Science Publishers, pp. 279–285.

[21] Namyst, R., and Méhaut, J. Madeleine : Une Interface de Communication Efficace pour les Environnements Multithreads. In *RenPar10, 10èmes Rencontres sur le Parallélisme* (Strasbourg, France, June 1998).

[22] Pakin, S., Lauria, M., and Chien, A. High Performance Messaging on Workstations : Illinois Fast Messages (FM) for Myrinet. In *Proc. of Supercomputing'95* (San Diego, California, December 1995). Available from *http ://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps*.

[23] Perez, C. Load balancing HPF programs by migrating virtual processors. In *Second International Workshop on High-Level Programming Models and Supportive Environments, HIPS'97* (April 1997), IEEE Computer Society Press.

[24] Prylli, L., and Tourancheau, B. BIP : A New Protocol designed for High-Performance Networking on Myrinet. In *Proc. of PC-NOW IPPS-SPDP98* (Orlando, USA, March 1998).

[25] Russell, R., and Hatcher, P. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing* (Atlanta, GA, February 1998). To appear.

[26] von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net : A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, December 1995).

[27] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. Active messages : a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture* (May 1992).

[28] Westrelin, R. Réseaux à haut débit et calcul parallèle. Master's thesis, Ecole Normale Supérieure de Lyon, June 1997.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399